# Python – Cours 1

Jean-Yves Thibon

Université Paris-Est Marne-la-Vallée

ESIPE – IR3

# Python

- Langage objet pour
  - prototypage ou développement rapide
  - programmation web
  - langage de contrôle de grandes applications
  - scripts, administration système
  - recherche et développement
  - sécurité (hacking ?)
  - et bien d'autres ...
- Extensible (C/C++/Fortran ou Java)
- Libre
- Mature (depuis 1991, communauté nombreuse et dynamique)
- Syntaxe concise, claire et cohérente

# CPython

- Comme son nom l'indique, écrit en C
- Extrêmement portable : Unix, Windows, MacOS, Symbian et autres systèmes embarqués
- Interprété/compilé (bytecode)
- Sûr (pas de pointeurs, gestion automatique de la mémoire)
- Nombreuses bibliothèques (réseau, bases de données, interfaces graphiques, son, vidéo ...)
- Extensible C/C++/Fortran (Cython)

# JPython (ex Jython)

- Totalement intégré avec Java
- Compilation directe en bytecode Java
- Importation directe des classes Java
- Héritage depuis les classes Java
- Support des JavaBeans, des applettes, des servlettes
- Même langage mais différences actuellement dans les modules d'extension

# Bref historique

- Première version en février 1991 (Guido van Rossum)
- Version 2.0 en 2000
- Aujourd'hui : version 2.7.5
- Fork :Version 3.0 en décembre 2008, (quelques incompatibilités avec les précédentes, 3.3.2 actuellement)
- Versions embarquées (téléphones portables : Nokia série 6, Palm ...)
- Stackless Python (microthreads, pour faciliter la programmation concurrente)
- Autres implémentations : IronPython (.NET), ActivePython, PyPy ...

Pour ce cours : **version 2 uniquement**

# Qui utilise Python ?

- Google ("Python were we can, C++ where we must")
- EDF (code Aster)
- Yahoo (Yahoo !mail)
- Microsoft (E-Commerce)
- Red Hat (Glint, setup tools), Ubuntu
- SGMLTools 2.0, LibreOffice (OpenOffice2.0),Vim
- Communauté scientifique (Sage ...)
- Et bien d'autres ...

# Caractéristiques du langage

- Tout est un objet
- Modules, classes, fonctions
- Exceptions
- Typage dynamique, polymorphisme
- Surcharge des opérateurs
- Syntaxe claire et intuitive : mix C/Algol, indentation pour délimiter les blocs (pseudo-code exécutable)

# Types de données

- Nombres : entiers, entiers longs, réels flottants, complexes
- Chaînes de caractères (immuables)
- Tuples, listes et dictionnaires (conteneurs)
- D'autres types natifs créés par des extensions en C (expressions régulières, descripteurs de fichiers, sockets...)
- Instances de classes définies en Python

# Exemples

Java :

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Python :

```
print "Hello, World!" # 3.0: print("Hello, World!")
```

# Plus convaincant : QuickSort en 2 lignes

```
def q(L):
  if len(L)<= 1: return L
  return q([x for x in L[1:] if x<L[0]])+[L[0]]+q([y for y in L[1:] if y>=L[0]])
```

```
>>> ll
[78, 46, 63, 20, 53, 10, 26, 52, 41, 54, 81, 75, 49, 21, 80, 60, 58, 56, 86,
40, 95, 92, 0, 4, 77, 12, 5, 59, 90, 57, 71, 3, 65, 27, 97, 89, 19, 38, 15, 85,
6, 62, 11, 33, 67, 61, 73, 44, 50, 17, 94, 48, 43, 34, 55, 24, 87, 70, 2, 16,
42, 25, 37, 68, 88, 30, 23, 7, 83, 74, 84, 39, 32, 98, 99, 22, 1, 45, 82, 69, 96,
31, 51, 91, 14, 13, 66, 36, 9, 18, 8, 79, 47, 72, 29, 76, 93, 64, 28, 35]
>>> q(ll)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
>>>
```

# Un script complet (commande Unix) I

```
#!/usr/bin/python
'''hello.py: le programme 'Hello world' bien connu.

Usage: hello.py [-h] [--help] [nom_de_personne]+
'''

import sys, getopt

def usage():
  'Print usage doc and exit.'
  print __doc__
  sys.exit()
```

# Un script complet (commande Unix) II

```
try:
  optlist, arglist = getopt.getopt(
    sys.argv[1:], 'h', ['help'])
except:
  usage()

for option in optlist:
  if option[0] in ['-h', '--help']:
    usage()
if arglist:
  for arg in arglist:
    print 'Hello %s!' % arg
else: print "Hello world!"
```

# Premiers pas I

On lance l'interpréteur en tapant `python` au prompt :

```
[jyt@liszt jyt]$ python
Python 2.6.1 (r261:67515, Dec 25 2008, 16:25:51)
[GCC 3.4.1 (Mandrakelinux 10.1 3.4.1-4mdk)] on linux2
Type "help", "copyright", "credits" or "license" for more info
>>> 2**100
1267650600228229401496703205376L
```

On peut aussi écrire des scripts

```
[jyt@liszt jyt]$ cat a.py
#!/usr/bin/env python
print 2**100
[jyt@liszt jyt]$ ./a.py
1267650600228229401496703205376
```

ou faire exécuter des fichiers

# Premiers pas II

```
[jyt@liszt jyt]$ cat b.py
print 2**100
[jyt@liszt jyt]$ python b.py
1267650600228229401496703205376
```

```
[jyt@liszt jyt]$ echo "print 2**100"|python
1267650600228229401496703205376
```

# Intepréteur : mode d'emploi I

Avec un langage interprété, on peut tester les instructions une à une avant de les incorporer dans le programme.
On aura donc un interpréteur de test dans une fenêtre, et un éditeur (vim, emacs, gedit ...) dans une autre.
La fonction `help(`*objet*`)` affiche l'aide en ligne d'un objet
On peut accéder à la liste des attributs d'un objet avec la commande `dir` :

```
>>> help(dir)
Help on built-in function dir in module __builtin__:

dir(...)
    dir([object]) -> list of strings

    If called without an argument, return the names in the current scope.
    Else, return an alphabetized list of names comprising (some of) the attributes
    of the given object, and of attributes reachable from it.
    If the object supplies a method named __dir__, it will be used; otherwise
    the default dir() logic is used and returns:
      for a module object: the module's attributes.
      for a class object:  its attributes, and recursively the attributes
        of its bases.
      for any other object: its attributes, its class's attributes, and
        recursively the attributes of its class's base classes.
```

# Intepréteur : mode d'emploi II

Par exemple, avec une chaîne de caractères, on obtient quelque chose du genre :

```
>>> dir("toto")
['__add__', '__class__', '__contains__', '__delattr__',
 '__doc__', ...
...
..., 'startswith', 'strip', 'swapcase', 'title', 'translate',
 'upper', 'zfill']
>>>
```

Le plus pratique pour débuter est l'IDE `idle`, développé en pur Python à titre de démonstration.
Il existe aussi un plugin Python (PyDev) pour Eclipse, et d'autres IDE du même genre (Eric).

# Conventions : essentiellement comme en C

```
>>> a=2
>>> a
2
>>> 1 == 2
False
>>> print "%s = %f" % ("pi", 3.1415926)
pi = 3.141593
```

Mais ...

# ... il y a un piège !

Les variables sont des références à des objets. Si `b = a`, une modification de `a` peut affecter `b` :

```
>>> a=2
>>> b=a
>>> a=3
>>> b
2          # jusqu'ici, RAS ...
>>> a=[1,2,3]
>>> b=a
>>> a[1]=5
>>> b
[1, 5, 3] # aïe aïe aïe !
```

# Solution : le module `copy`

Pas gênant si on est conscient du problème. Au passage : la bibliothèque se compose de *modules* que l'on doit *importer*.

```
>>> help('copy')
>>> import copy
>>> x=[1,2,3]
>>> y=copy.copy(x)
>>> y
[1, 2, 3]
>>> x[1]=5
>>> x
[1, 5, 3]
>>> y
[1, 2, 3]
```

# Pour les objets complexes, utiliser `deepcopy`

```
>>> u=[1,[2,3],4]
>>> v=copy.copy(u)
>>> u[0]=6
>>> u
[6, [2, 3], 4]
>>> v
[1, [2, 3], 4]
>>> u[1][0]=7
>>> u
[6, [7, 3], 4]
>>> v
[1, [7, 3], 4]
>>> p=[1, [2,[3,4],5], [6,7]]
>>> q=copy.deepcopy(p)
>>> p[1][1][0]=8
>>> p
[1, [2, [8, 4], 5], [6, 7]]
>>> q
[1, [2, [3, 4], 5], [6, 7]]
```

# Blocs et indentation I

Pas d'accolades ou de `begin/end`. C'est l'indentation qui détermine la structure des blocs :

```
>>> for i in range(0,256):
...   if chr(i).isalnum():
...     print i, chr(i)
...   else: pass
```

Le point-virgule en fin de ligne est optionnel. On peut l'utiliser pour mettre plusieurs instructions par ligne :

```
>>> x='abra'; y='cadabra'; z=x+y; print z
abracadabra
```

On peut utiliser des \ pour continuer une instruction sur la ligne suivante :

```
print 2+3*4/27% \
42+37
```

Il est parfois plus simple (et recommandé) d'utiliser des parenthèses :

# Blocs et indentation II

```
print (2+3*4/27
 %42+37)
```

On ne doit pas mélanger espaces et tabulations. La convention la plus répandue est d'indenter de 4 espaces.

# Structures de contrôle I

Noter la présence des deux points " :" après `if, else, for, while`,...

```
if condition:
    faire_si_vrai()
elif autre_condition:
    faire_autre_truc()
else:
    faire_si_tout_faux()

while condition:
    iteration()
    if fini: break      # sort de la boucle
    if pouet: continue # saute ce qui suit
    autre_chose()
```

Les itérations portent sur des listes, ou plus généralement sur des objets itérables :

# Structures de contrôle II

```
for element in liste:
    manip(element)
    if foo(element): break
    if quux(element): continue
    chose(element)
```

```
>>> range(0,10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for i in range(0,10): print i**2
```

Pour itérer sur des entiers on engendre leur liste au moyen de la fonction `range()`
`range`([debut=0],fin,[pas=1])

# Les types : entiers et flottants I

```
>>> 2+3*4
14
>>> 1<<8     # aussi >>, &, |, ^ (XOR)
256
>>> 5**3
125
>>> 42%10
2
>>> 2**(1/12)
1
>>> 1/12     # attention à la division entière
0
>>> 2**(1.0/12)
1.059463094359295 3
>>> 2**200
1606938044258990275541962092341162602522202993782792835301376L
```

On voit que Python supporte la multiprécision (utile pour la cryptographie, entre autres).

# Les types : entiers et flottants II

Les fonctions mathématiques de base sont dans le module `math` :

```
>>> import math
>>> math.cos(math.pi/3)
0.50000000000000011
```

On dispose de fonctions de conversion entre bases

```
>>> int("0256")
256
>>> int("1101101",2)
109
>>> hex(_)
'0x6d'
>>> int(_,16)
109
>>> bin(109)
'0b1101101'
```

# Les types : entiers et flottants III

Pour les flottants en multiprécision, c'est le module `decimal` qu'il faut utiliser :

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN,
        Emin=-999999999, Emax=999999999,
        capitals=1, flags=[],
        traps=[Overflow, InvalidOperation, DivisionByZero])
>>> getcontext().prec=100
>>> Decimal(2).sqrt()
Decimal("1.41421356237309504880168872420969807856967187537694
              07317667973799073247846210703885038753432764157
>>>
```

# Les types : les chaînes I

On peut utiliser deux types de guillemets. `'toto'` et `"toto"` sont équivalentes. Permet d'écrire `"l'apostrophe"` ou `'les "guillemets"'`. Les triples guillemets `"""..."""` ou `"'...'"` permettent d'écrire sur plusieurs lignes. Une chaîne qui traîne au milieu d'un programme est vue comme un commentaire.

```
>>> "abc"+"def"      # concatenation avec +
'abcdef'
>>> "abc"*2
'abcabc'
>>> "abc%sdef%d" % ("toto",4) # formatage style "printf"
'abctotodef4'
>>> "%08X" % 42069    # y compris les conversions de base
'0000A455'
>>> "%%%s" % "d" % 12 # On peut écrire du code illisible
12
>>> "rac" in "abracadabra"
True
>>> 'abracadabra'.__contains__('rac')  # (équivalent)
```

Les chaines ont de nombreuses méthodes très pratiques (essayer `dir("")`).

# Les types : les chaînes II

```
>>> len("toto")
4
>>> "toto".upper()
'TOTO'
>>> "      x yz    ".strip()
'x yz'
>>> "albert".startswith("al")
True
>>> "salamalec".replace("al","ila")
'silaamilaec'
>>> "l'an de l'ananas".count("an")
3
>>> "     piano\n".strip()
'piano'
```

Le module `string` offre quelques fonctionalités supplémentaires.

# Les types : tuples et listes I

En Python, liste=tableau. Les indices commencent à 0.

```
>>> ll = ['toto', 42, ['ga', 'bu', 'zo', 'meu'], 999]
>>> ll[2][0]
'ga'
>>> ll.append('foo')
>>> ll
['toto', 42, ['ga', 'bu', 'zo', 'meu'], 999, 'foo']
>>> ll.pop()
'foo'
>>> [1,2,4,8]+[16,32]
[1, 2, 4, 8, 16, 32]
>>> liste=["toto",42,True]
>>> liste[0]=123
>>> liste
[123, 42, True]
>>> 123 in liste
True
>>> 123 not in liste
False
```

# Les types : tuples et listes II

Les tuples sont définis par des parenthèses. Ils ont essentiellement les mêmes méthodes mais ils ne sont pas *mutables*

```
>>> (1,2,3)*2
(1, 2, 3, 1, 2, 3)
>>> pasliste=(1, False)
>>> pasliste[0]
1
>>> pasliste[0]=69
TypeError: object doesn't support item assignment
>>> list(pasliste)
[1, False]
>>> range(4)
[0, 1, 2, 3]
>>> tuple(range(4))
(0, 1, 2, 3)
>>> 1,2,3      # la virgule construit des tuples
(1, 2, 3)


>>> s='abc'
```

# Les types : tuples et listes III

```
>>> list(s)
['a', 'b', 'c']
>>> tuple(s)
('a', 'b', 'c')
>>> for x in s: print x
...
a
b
c
>>> "aa".join( ('d','e','f') )
'daaeaaf'
>>> "aa".join( ['d','e','f'] )
'daaeaaf'
>>> "aa".join('def')
'daaeaaf'
>>> "abcdefgihj"[4]
'e'
>>> "abcdefghij"[4:7]  # un morceau
'efg'
```

# Les types : tuples et listes IV

```
>>> [1,2,4,8,16][2:]    # de 2 à la fin
[4, 8, 16]
>>> s="ga bu zo meu"
>>> s[-3:]              # indice -1 = le dernier
'meu'
>>> s = s.split(" ")
>>> s
['ga', 'bu', 'zo', 'meu']
>>> "_".join([x*2 for x in s])
'gaga_bubu_zozo_meumeu'
```

# Les types : dictionnaires I

Les dictionnaires sont des tables de hachage, définies par des items <clef> :<valeur> entre accolades. Les clefs doivent être des objets non mutables : entiers, chaînes, tuples.

```
>>> bazar={"e": 2.71828, "jaune": (255,255,0), "vrai": True,
10: "dix", "liste": [4, 2, {False: "faux"}], (0,0):"origine"}
>>> bazar["liste"]
[4, 2, {False: "faux"}]
>>> bazar["liste"][2][False]
'faux'
>>> bazar.has_key("truc")
False
>>> bazar["truc"]="chose"
>>> "truc" in bazar
True
>>> del bazar["truc"]
>>> dict([("k1",10),("k2","v")]) # construction à partir d'une
                                 # liste de couples
{'k2': 'v', 'k1': 10}
>>> _.items()                    # et inversement
[('k2', 'v'), ('k1', 10)]
```

# Les types : dictionnaires II

```
>>> for k,v in bazar.items(): print k, '=', v
...
jaune = (255, 255, 0)
(0, 0) = origine
e = 2.71828
10 = dix
liste = [4, 2, {False: 'faux'}]
vrai = True
>>> t='Le cheval de mon cousin ne mange du foin que le dimanch
>>> d = {}
>>> for x in t:
...     if d.has_key(x): d[x] += 1
...     else: d[x] = 1
...
>>> d
{'a': 3, ' ': 11, 'c': 3, 'e': 8, 'd': 3, 'g': 1, 'f': 1,
 'i': 3, 'h': 2, 'm': 3, 'L': 1, 'o': 3, 'l': 2, 'q': 1,
 's': 1, 'u': 3, 'v': 1, 'n': 6}
```

# Les types : dictionnaires III

Formatage par dictionnaire :

```
html = """
<TITLE> Inscription </TITLE>
<H1> Inscription terminée</H1>
<H2> Fédération Française de Friture à Froid </H2>
<H2> 42ème Congrès </H2>
<BR>
<BR> %(nom)s %(prenom)s
<BR> %(email)s
<HR>"""

data = {'nom':'Cussonnet', 'prenom':'Simon',
   'email':'simon.cussonnet@thorel.eur'}



print html % data

<TITLE> Inscription </TITLE>
```

# Les types : dictionnaires IV

```
<H1> Inscription terminée</H1>
<H2> Fédération Française de Friture à Froid </H2>
<H2> 42ème Congrès </H2>
<BR>
<BR> Cussonnet Simon
<BR> simon.cussonnet@thorel.eur
<HR>

# Alternative :
from string import Template
s = Template('$qui ne mange du $quoi que $quand')
t = s.substitute(qui='Le cheval de mon cousin', quoi='du foin'
                 quand='le dimanche.')
print t
```

# Le typage en Python I

Typage dynamique : les types sont implicitement *polymorphes*
On peut surcharger les opérateurs
*Duck typing* : Connaître le type d'un objet n'a pas d'importance, il faut seulement s'assurer qu'on peut lui appliquer les traitements souhaités
"si ça marche comme un canard et si ça cancane comme un canard, alors ce doit être un canard (et si c'est une oie, on doit pouvoir faire avec)".
Utiliser la coertion si un objet doit être d'un type particulier (`str(x)` plutôt que de demander `isinstance(x,str)`.
EAFP : "It's easier to ask forgiveness than permission". Utiliser le mécanisme de gestion des exceptions (`try ... except`) pour déterminer le traitement approprié.

# Le typage en Python II

## Polymorphisme :

```
>>> def add(a,b): return a+b
>>> add(3,5)
8
>>> add(3,5.0)
8.0
>>> add("abra","cadabra")
'abracadabra'
>>> add([1,2],[3,4])
[1, 2, 3, 4]
>>>
```

# Le typage en Python III

```
>>> print type(5), type(5.0), type("abra"),type([1,2])
<type 'int'> <type 'float'> <type 'str'> <type 'list'>
>>> 2+"2"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and
>>> 2*"2"
'22'
```

# Les fonctions I

## Mots clefs : `def ... return`

```
def f():
    return None

def cube(x):
    return x**3

def aire(x1,y1,x2,y2):
    delta_x=x2-x1
    delta_y=y2-y1
    return abs(delta_x*delta_y)

def aire(a,b,x2=None,y2=None):
    if x2: x1,y1=a,b
    else: (x1,y1),(x2,y2)=a,b
    return abs((x2-x1)*(y2-y1))
```

# Les fonctions II

Le fonctions peuvent avoir des argument nommés avec des valeurs par défaut.
La deuxième version marche avec `aire(1,2,3,4)` mais aussi `aire((1,2),(3,4))`
On notera le test `if x2` :
`False, 0, None, [], {}, ""` ont une valeur booleenne "Faux", les autres objets sont "vrais".
On observera aussi le polymorphisme en action :
`a, b` peuvent être des nombres ou des couples.

```
def wget(host,port=80,uri="/"):
  return urllib.urlopen("http://%s:%d%s" %
                        (host,port,uri)).read()

wget("google.com")
wget("google.com",80,"/search?q=perdu")
wget("google.com",uri="/search?q=perdu")
wget(uri="/search?q=perdu",host="google.com")
```

Interdit : `fonction(param=valeur,autreparam)`

# Les fonctions III

On peut avoir un nombre variable d'arguments

```
def printf(format_string, *args):
    print format_string % args

printf("%s: %d","trois",3)

liste=["riri", "fifi", "loulou", 17]
printf("%s + %s + %s = %d",*liste)

toto(a,b,c) = toto(*[a,b,c])
```

L'opréateur $*$ extrait les éléments d'une liste (seulement dans un appel de fonction).

# Les fonctions IV

On peut aussi avoir des argument nommés arbitraires

```
def apply_font(font="Helvetica", size=12, **styles):
    print "Font name is %s."%font
    print "Font size is %d."%size
    print "Extra attributes:"
    for attrname,attrvalue in styles.items():
        print "\t%s: %s"%(attrname,str(attrvalue))

>>> apply_font(size=24,color="red",background="black")
Font name is Helvetica.
Font size is 24.
Extra attributes:
    background: black
    color: red
>>> apply_font(**{"color": "red", "background": "black"})
```

L'opérateur $**$ extrait les arguments d'un dictionnaire.

# Les fonctions V

Fonctions anonymes : `lambda` vient du LISP, s'utilise avec `map, filter, reduce`.

```
>>> compose=lambda f,g:lambda x:g(f(x))
>>> plus4fois2=compose(lambda x:x+4,lambda x:x*2)
>>> plus4fois2(5)
18

>>> map(lambda x: x*x, [1,2,3,4])
[1, 4, 9, 16]
>>> filter(lambda x: chr(x).isdigit(), range(0,256))
[48, 49, 50, 51, 52, 53, 54, 55, 56, 57]

>>> def add(*ll):
reduce(lambda x,y:x+y,ll)

>>> add(*range(100))
4950
```

# Compréhensions de listes I

Syntaxe très puissante empruntéee à Haskell. La préférer à `map-lambda - filter -reduce`

```
>>> [x*x for x in [1,2,3,4]]
[1, 4, 9, 16]

>>> [x for x in range(0,256) if chr(x).isdigit()]
[48, 49, 50, 51, 52, 53, 54, 55, 56, 57]

>>> noprimes = [j for i in range(2, 8)
                  for j in range(i*2, 50, i)]
>>> primes = [x for x in range(2, 50) if x not in noprimes]
>>> print primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

# Compréhensions de listes II

Exemple : extraire en une seule instruction les variables affectées dans un fichier de configuration

```
>>> d=dict([kv.split("=",1)
      for kv in open("/etc/libuser.conf").read().split("\n")
      if "=" in kv and not kv.strip().startswith("#")])
>>> d
{'LU_GIDNUMBER ': ' 500', 'mailspooldir ': ' /var/mail',
  'moduledir ': ' /usr/lib/libuser',
  'modules ': ' files shadow', 'skeleton ': ' /etc/skel',
  'create_modules ': ' files shadow', 'LU_UIDNUMBER ': ' 500',
  'LU_USERNAME ': ' %n', 'LU_GROUPNAME ': ' %n',
   'crypt_style ': ' md5'}
```

On notera au passage qu'on ouvre un fichier avec `open`. Renvoie un `file object` dont la méthode `read` lit tout le contenu sous forme d'une chaîne.

# Orientation objet : les classes et leurs instances I

- Tout est un objet (entiers, fonctions, modules, sockets ...)
- Type structurés : classes et instances de classes
- On accède aux attributs et aux méthodes d'une classe avec `dir()`
- En python 2.x : "old" and "new" styles
- Les classes *newstyle* héritent du type `object`
- Syntaxe : `class Toto:` ou `class Toto(object):`

# Orientation objet : les classes et leurs instances II

```
>>> class Toto:pass # old style

>>> a=Toto()          # Création d'une instance
>>> dir(a)
['__doc__', '__module__']a    # Pas grand-chose a voir
>>> class Titi(object): pass # new style, dérive de object

>>> b=Titi()
>>> dir(b)      # Là, il y a du monde ...
['__class__', '__delattr__', '__dict__', '__doc__',
'__format__', '__getattribute__', '__hash__', '__init__',
'__module__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__']
>>>
```

Les attributs commençant et finissant par deux soulignés sont les *méthodes spéciales*.

# Orientation objet : les classes et leurs instances III

Héritage :

```
>>> class Cbase: pass
>>> class Cderivee(Cbase): pass
>>> Ibase=CBase() ; Iderivee=Cderivee()
>>> isinstance(Ibase,Cbase)
True
>>> isinstance(Iderivee,Cbase)
True
>>> isinstance(Ibase,Cderivee)
False
```

Les classes sont "callable", ce qui n'est pas le cas de tous les objets

```
>>> callable(42), callable(Cbase)
(False, True)
```

# Orientation objet : les classes et leurs instances IV

La méthode spéciale `__init__` est applée imédiatement après la création d'une instance. Son premier argument est l'instance elle même, conventionnellement appelée `self`, les suivants sont les paramètres éventuels :

```
class defaultdict(dict):
""" Renvoie une valeur par défaut
    si une clef n'est pas affectée """

    def __init__(self, default=None):
        dict.__init__(self)
        self.default = default

    def __getitem__(self, key):
        try:
            return dict.__getitem__(self, key)
        except KeyError:
            return self.default
```

# Orientation objet : les classes et leurs instances V

On notera l'usage de `try ... except` plutôt qu'un test `if key in self`. Résultat :

```
>>> d = defaultdict(0.0)
>>> e={1:4.5, 2:7.8}
>>> d.update(e)
>>> d
{1: 4.5, 2: 7.8}
>>> d[3]
0.0
>>>
```

# Orientation objet : les classes et leurs instances VI

Les classes "newstyle" possèdent une méthode `__new__` qui prend en charge la construction de l'instance. Elle est utile pour sous-classer les types non mutables :

```
class CapString(str):
    def  __new__(cls,s):
        return  str.__new__(cls,s.lower().capitalize())

    def __add__(self,x):
        return CapString (str.__add__(self,x.lower()))

x=CapString("toTO")
y = CapString("tITi")
z = x+y
print x,y,z
print type(x), type(y), type(z)
>>>
Toto Titi Tototiti
<class '__main__.CapString'> <class '__main__.CapString'>
          <class '__main__.CapString'>
```